CIS 550 Project Report

Datalake Management System

Team of:

Marvin Liu
Yimeng Xu
Shravan Chopra
Sujay Suresh Kumar

Introduction:

A **data lake** is a large-scale storage repository and processing engine. A data lake provides "massive storage for any kind of data, enormous processing power and the ability to handle virtually limitless concurrent tasks or jobs". By being able to access a huge amount of data, users have the ability to obtain a large amount of information. Because many users can access this data lake, creating permissions and holding all the data from all these users can cause issues in performance and correctly stored information. The datalake management system must be able to quickly obtain whatever data the user requests. Therefore, it might be efficient to use indexing, sorting, or hashing for query optimization so that the user can be quickly provided with the correct information. However, there are many other ways to speed up the performance. It is also possible to have certain data to be accessible to certain users, and therefore, it is possible to access the same pieces of data at the same time (or at least at similar times). When accessing the same pieces of data at once, the datalake management system must ensure that there are no read-write, write-read, or write-write possibilities between users if the file is writable.

Architecture and Roadmap (The main components):

Backend Design:

- The system uses Amazon S3 to store the files (.json, .csv, .xml, .pdf, .txt, .docx)
- We used S3 for this because S3 uses key value store which makes it easy for us to store any data with a key and query the data(file in this case) using the key.
- The system uses MongoDB to store intermediate collections that are required to process the linking mechanism and generate the graph
- The MongoDB collections are as follows:
  - Inverted Index (‘InvertedIndex’)
  - Forward index (‘node’)
  - Converted JSON with node ids (‘testCollection’)
- The system uses MySQL to store 2 tables for the following
  - username and password data
  - Permission to the documents
- The system uses the Dracula graph library to render graphical outputs of word searches.

Frontend Design:

- Node.js Express is used to write and run our application
- EJS is used to hold our client side code because we are using Node.js Express

- EJS is CanJS's default template language, which provides live binding when used with Observes. The reason to use EJS rather than static HTML is that EJS provide access from server to client.
- The Node.js Express app is hosted on Amazon EC2

Implementation:

Uploading Files:

When a file is uploaded at the client side, the user gives the name of the file and submits the file for upload by clicking a button on the front end side. The file, at the server, is downloaded to a temporary local directory “data” and stored with the filename specified by the user. This file will then be uploaded to Amazon S3 to join the rest of the files that already exist in the data lake.

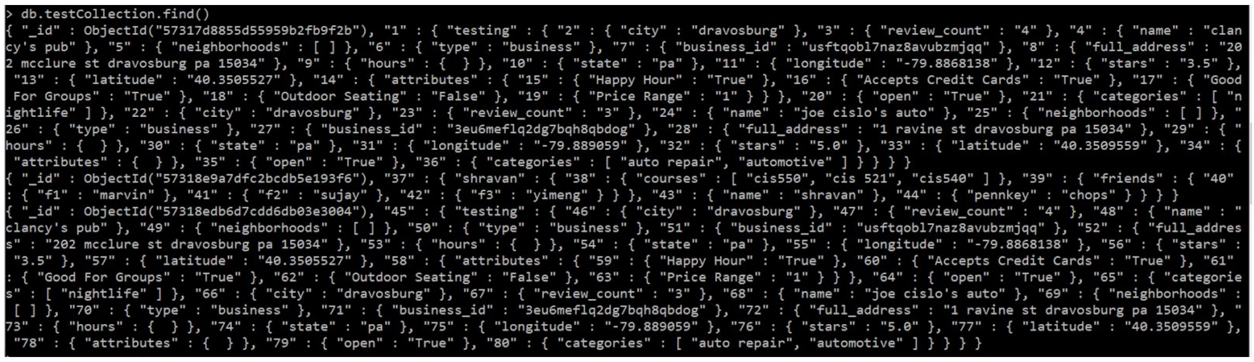
Converting JSON:

Once the file is uploaded, we parse the file and convert the data in the file into a json file with node id’s, into order to store into mongo as a key value pair for processing the inverted index and forward index. The parsing is done based on the file extension. A python script parses json files and slaps a node to every key value pair in the json file. An example below shows the result of the conversion
Original JSON file:
{“a” : “b” , “c” : [“d” ,”e” ,”f”] , “g” : { “h” : “i” , “j” : “k” , “l” : { “m” : “n” , “o” : “p”} }}

Converted JSON with node ids:
{“1” : { filename :
{“2” : {“a” : “b”}, “3”: {“c” : [“d”, “e”, “f”]}, “4” : { “g” : { “5” : {“h” : “i”}, “6”:{“j” : ”k”}, 7: {“l” : { 8 :{“m”:”n”}, 9 : {“o” : “p”}}}}}
}}

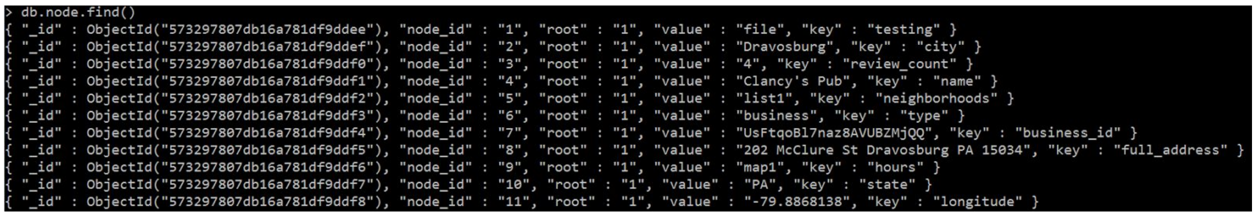
If the file is a csv file, or an xml file, it is converted to an intermediate json file that preserves the hierarchy of the data and then passed to the json parser as mentioned above. The node id’s are global and incremental, i.e, if the node id’s end at 10 for one document, the next document will have a starting id of 11. Notice that the first node id in the converted json file maps to the filename. This node id corresponds to the document id which helps in looking up the document and permissions during the linking part. Once the json is created, it is pushed into a mongo collection ‘testCollection’.

For each of the node ids that exist, there is a corresponding key, value, and root. As we are pushing the converted json onto ‘testCollection’, we are also pushing these corresponding node ids and their corresponding key, value, and root into the forward index mongo collection ‘node’. Each document uploaded into the server (data directory) has an entry in the testCollection collection on mongo. Each node id, key, value, root tuple in this document also has an entry in the ‘node’ collection on mongo. An example of the testCollection would be:



```
> db.testCollection.find()
{ "_id" : ObjectId("57317d8855d55959b2fb9f2b"), "1" : { "testing" : { "2" : { "city" : "dravosburg" }, "3" : { "review_count" : "4" }, "4" : { "name" : "clan
cy's pub" }, "5" : { "neighborhoods" : [ ] }, "6" : { "type" : "business" }, "7" : { "business_id" : "usftqobl7naz8avubzmjqq" }, "8" : { "full_address" : "20
2 mcclure st dravosburg pa 15034" }, "9" : { "hours" : {  } }, "10" : { "state" : "pa" }, "11" : { "longitude" : "-79.8868138" }, "12" : { "stars" : "3.5" },
 "13" : { "latitude" : "40.3505527" }, "14" : { "attributes" : { "15" : { "Happy Hour" : "True" }, "16" : { "Accepts Credit Cards" : "True" }, "17" : { "Good
 For Groups" : "True" }, "18" : { "Outdoor Seating" : "False" }, "19" : { "Price Range" : "1" } } }, "20" : { "open" : "True" }, "21" : { "categories" : [ "n
ightlife" ] }, "22" : { "city" : "dravosburg" }, "23" : { "review_count" : "3" }, "24" : { "name" : "joe cislo's auto" }, "25" : { "neighborhoods" : [ ] }, "
26" : { "type" : "business" }, "27" : { "business_id" : "3eu6meflq2dg7bqh8qbdog" }, "28" : { "full_address" : "1 ravine st dravosburg pa 15034" }, "29" : { "
hours" : {  } }, "30" : { "state" : "pa" }, "31" : { "longitude" : "-79.889059" }, "32" : { "stars" : "5.0" }, "33" : { "latitude" : "40.3509559" }, "34" : {
 "attributes" : {  } }, "35" : { "open" : "True" }, "36" : { "categories" : [ "auto repair", "automotive" ] } } } }
{ "_id" : ObjectId("57318e9a7dfc2bcdb5e193f6"), "37" : { "shravan" : { "38" : { "courses" : [ "cis550", "cis 521", "cis540" ] }, "39" : { "friends" : { "40"
: { "f1" : "marvin" }, "41" : { "f2" : "sujay" }, "42" : { "f3" : "yimeng" } } }, "43" : { "name" : "shravan" }, "44" : { "pennkey" : "chops" } } } }
{ "_id" : ObjectId("57318edb6d7cdd6db03e3004"), "45" : { "testing" : { "46" : { "city" : "dravosburg" }, "47" : { "review_count" : "4" }, "48" : { "name" : "
clancy's pub" }, "49" : { "neighborhoods" : [ ] }, "50" : { "type" : "business" }, "51" : { "business_id" : "usftqobl7naz8avubzmjqq" }, "52" : { "full_addres
s" : "202 mcclure st dravosburg pa 15034" }, "53" : { "hours" : {  } }, "54" : { "state" : "pa" }, "55" : { "longitude" : "-79.8868138" }, "56" : { "stars" :
 "3.5" }, "57" : { "latitude" : "40.3505527" }, "58" : { "attributes" : { "59" : { "Happy Hour" : "True" }, "60" : { "Accepts Credit Cards" : "True" }, "61"
: { "Good For Groups" : "True" }, "62" : { "Outdoor Seating" : "False" }, "63" : { "Price Range" : "1" } } }, "64" : { "open" : "True" }, "65" : { "categorie
s" : [ "nightlife" ] }, "66" : { "city" : "dravosburg" }, "67" : { "review_count" : "3" }, "68" : { "name" : "joe cislo's auto" }, "69" : { "neighborhoods" :
 [ ] }, "70" : { "type" : "business" }, "71" : { "business_id" : "3eu6meflq2dg7bqh8qbdog" }, "72" : { "full_address" : "1 ravine st dravosburg pa 15034" }, "
73" : { "hours" : {  } }, "74" : { "state" : "pa" }, "75" : { "longitude" : "-79.889059" }, "76" : { "stars" : "5.0" }, "77" : { "latitude" : "40.3509559" },
 "78" : { "attributes" : {  } }, "79" : { "open" : "True" }, "80" : { "categories" : [ "auto repair", "automotive" ] } } } }
```

An example of the preliminary ‘node’ collection would be:

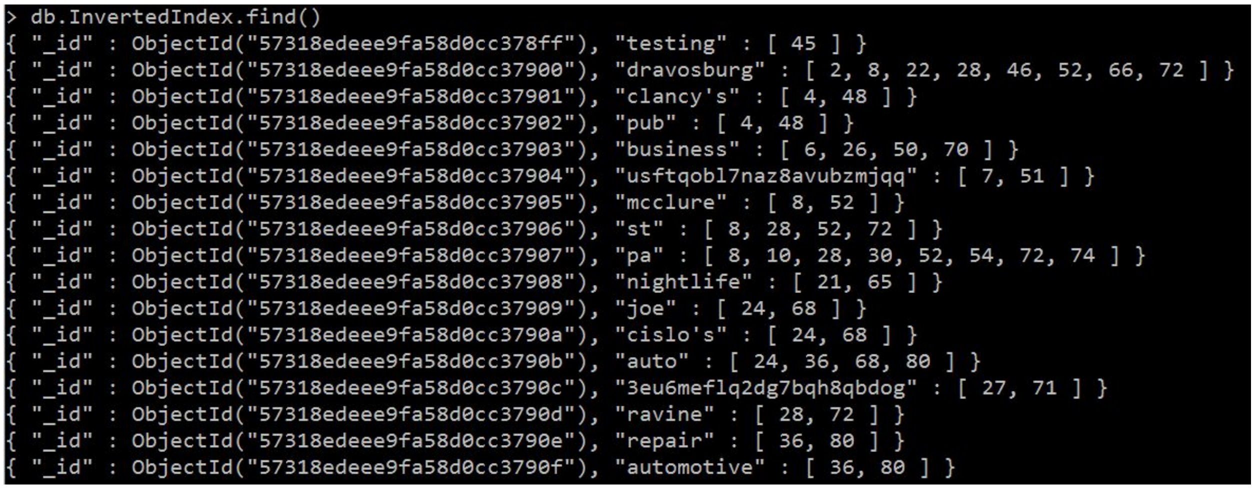


```
> db.node.find()
{ "_id" : ObjectId("573297807db16a781df9ddee"), "node_id" : "1", "root" : "1", "value" : "file", "key" : "testing" }
{ "_id" : ObjectId("573297807db16a781df9ddef"), "node_id" : "2", "root" : "1", "value" : "Dravosburg", "key" : "city" }
{ "_id" : ObjectId("573297807db16a781df9ddf0"), "node_id" : "3", "root" : "1", "value" : "4", "key" : "review_count" }
{ "_id" : ObjectId("573297807db16a781df9ddf1"), "node_id" : "4", "root" : "1", "value" : "Clancy's Pub", "key" : "name" }
{ "_id" : ObjectId("573297807db16a781df9ddf2"), "node_id" : "5", "root" : "1", "value" : "list1", "key" : "neighborhoods" }
{ "_id" : ObjectId("573297807db16a781df9ddf3"), "node_id" : "6", "root" : "1", "value" : "business", "key" : "type" }
{ "_id" : ObjectId("573297807db16a781df9ddf4"), "node_id" : "7", "root" : "1", "value" : "UsFtqoBl7naz8AVUBZMjQQ", "key" : "business_id" }
{ "_id" : ObjectId("573297807db16a781df9ddf5"), "node_id" : "8", "root" : "1", "value" : "202 McClure St Dravosburg PA 15034", "key" : "full_address" }
{ "_id" : ObjectId("573297807db16a781df9ddf6"), "node_id" : "9", "root" : "1", "value" : "map1", "key" : "hours" }
{ "_id" : ObjectId("573297807db16a781df9ddf7"), "node_id" : "10", "root" : "1", "value" : "PA", "key" : "state" }
{ "_id" : ObjectId("573297807db16a781df9ddf8"), "node_id" : "11", "root" : "1", "value" : "-79.8868138", "key" : "longitude" }
```

Once the converted json with node ids is uploaded into mongo, the process of generating the inverted index and completing the forward index can begin.

Inverted Index:

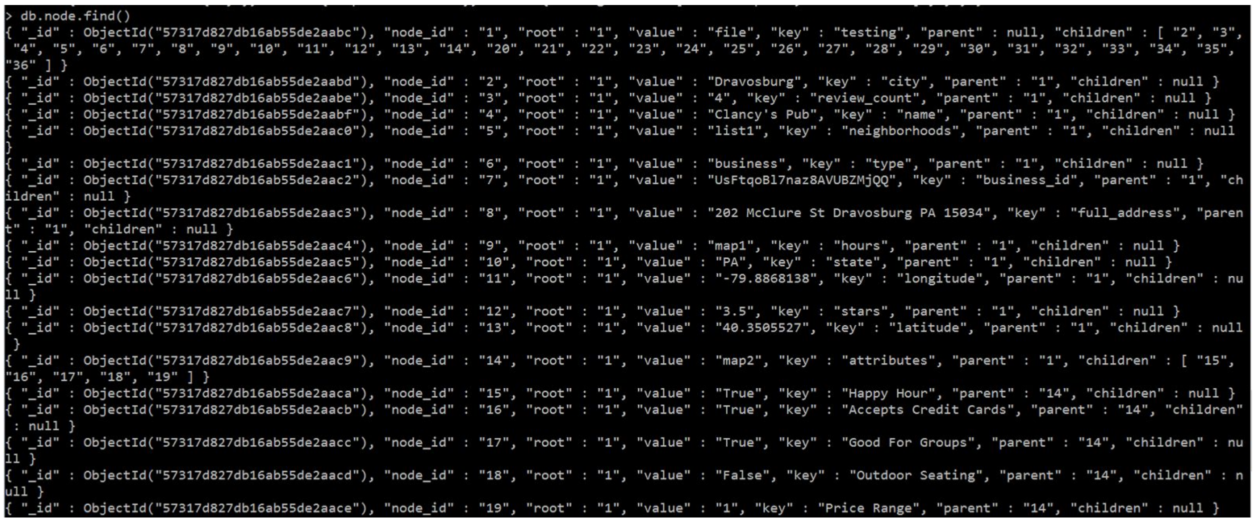
The inverted index is created by using the node ids from the converted json. It takes each unique word and finds which node ids they correspond to. To find these unique words the common words that were in the list of stop words from http://xpo6.com/list-of-english-stop-words/, punctuation, and any numbers were removed. All words in the documents that were uploaded are also converted so that they will be made up of lowercase letters. By doing this, we will be able to ensure that “this” and “This” are the same word. Because some values in the document can be a large string collection of words, the values that are string are split on spaces so that the node id that this large string is mapped to will map to each individual word in the string. When each word is normalized, the words and list of node ids are mapped to their corresponding values. This essentially creates a list of node ids that map to each unique word. This data is pushed into a mongo collection ‘InvertedIndex’. In order to ensure that this list of node ids does not have duplicates, the corresponding node id is appended to that list instead of creating another document in the collection. An example would be:



```
> db.InvertedIndex.find()
{ "_id" : ObjectId("57318edeee9fa58d0cc378ff"), "testing" : [ 45 ] }
{ "_id" : ObjectId("57318edeee9fa58d0cc37900"), "dravosburg" : [ 2, 8, 22, 28, 46, 52, 66, 72 ] }
{ "_id" : ObjectId("57318edeee9fa58d0cc37901"), "clancy's" : [ 4, 48 ] }
{ "_id" : ObjectId("57318edeee9fa58d0cc37902"), "pub" : [ 4, 48 ] }
{ "_id" : ObjectId("57318edeee9fa58d0cc37903"), "business" : [ 6, 26, 50, 70 ] }
{ "_id" : ObjectId("57318edeee9fa58d0cc37904"), "usftqobl7naz8avubzmjqq" : [ 7, 51 ] }
{ "_id" : ObjectId("57318edeee9fa58d0cc37905"), "mcclure" : [ 8, 52 ] }
{ "_id" : ObjectId("57318edeee9fa58d0cc37906"), "st" : [ 8, 28, 52, 72 ] }
{ "_id" : ObjectId("57318edeee9fa58d0cc37907"), "pa" : [ 8, 10, 28, 30, 52, 54, 72, 74 ] }
{ "_id" : ObjectId("57318edeee9fa58d0cc37908"), "nightlife" : [ 21, 65 ] }
{ "_id" : ObjectId("57318edeee9fa58d0cc37909"), "joe" : [ 24, 68 ] }
{ "_id" : ObjectId("57318edeee9fa58d0cc3790a"), "cislo's" : [ 24, 68 ] }
{ "_id" : ObjectId("57318edeee9fa58d0cc3790b"), "auto" : [ 24, 36, 68, 80 ] }
{ "_id" : ObjectId("57318edeee9fa58d0cc3790c"), "3eu6meflq2dg7bqh8qbdog" : [ 27, 71 ] }
{ "_id" : ObjectId("57318edeee9fa58d0cc3790d"), "ravine" : [ 28, 72 ] }
{ "_id" : ObjectId("57318edeee9fa58d0cc3790e"), "repair" : [ 36, 80 ] }
{ "_id" : ObjectId("57318edeee9fa58d0cc3790f"), "automotive" : [ 36, 80 ] }
```

Forward Index:

The forward index holds the node id, root id, parent id, children ids, the node id, and the key of the node id. The root id, key, value, and node id are first found when the original JSON is converted into one with node ids. Then, we add the parent and children ids when we create the inverted index. The reason for this is because it is more efficient to obtain the root id, key, value, and node id when converting the JSON file since we are already obtaining these values. If the node id is the one that maps to the filename, this is the root id. This root id will have one or more children ids but will have a null parent id. The nodes of the graph that are leaves will have a null children id. All other nodes will have a parent id value and one or more children id values. Because of the way the forward index is made, we can easily create a graph and do searches through any data lake. This data is pushed into a mongo collection 'node'. An example would be:



```
> db.node.find()
{ "_id" : ObjectId("57317d827db16ab55de2aabc"), "node_id" : "1", "root" : "1", "value" : "file", "key" : "testing", "parent" : null, "children" : [ "2", "3",
 "4", "5", "6", "7", "8", "9", "10", "11", "12", "13", "14", "20", "21", "22", "23", "24", "25", "26", "27", "28", "29", "30", "31", "32", "33", "34", "35",
"36" ] }
{ "_id" : ObjectId("57317d827db16ab55de2aabd"), "node_id" : "2", "root" : "1", "value" : "Dravosburg", "key" : "city", "parent" : "1", "children" : null }
{ "_id" : ObjectId("57317d827db16ab55de2aabe"), "node_id" : "3", "root" : "1", "value" : "4", "key" : "review_count", "parent" : "1", "children" : null }
{ "_id" : ObjectId("57317d827db16ab55de2aabf"), "node_id" : "4", "root" : "1", "value" : "Clancy's Pub", "key" : "name", "parent" : "1", "children" : null }
{ "_id" : ObjectId("57317d827db16ab55de2aac0"), "node_id" : "5", "root" : "1", "value" : "list1", "key" : "neighborhoods", "parent" : "1", "children" : null
}
{ "_id" : ObjectId("57317d827db16ab55de2aac1"), "node_id" : "6", "root" : "1", "value" : "business", "key" : "type", "parent" : "1", "children" : null }
{ "_id" : ObjectId("57317d827db16ab55de2aac2"), "node_id" : "7", "root" : "1", "value" : "UsFtqoBl7naz8AVUBZMjQQ", "key" : "business_id", "parent" : "1", "ch
ildren" : null }
{ "_id" : ObjectId("57317d827db16ab55de2aac3"), "node_id" : "8", "root" : "1", "value" : "202 McClure St Dravosburg PA 15034", "key" : "full_address", "paren
t" : "1", "children" : null }
{ "_id" : ObjectId("57317d827db16ab55de2aac4"), "node_id" : "9", "root" : "1", "value" : "map1", "key" : "hours", "parent" : "1", "children" : null }
{ "_id" : ObjectId("57317d827db16ab55de2aac5"), "node_id" : "10", "root" : "1", "value" : "PA", "key" : "state", "parent" : "1", "children" : null }
{ "_id" : ObjectId("57317d827db16ab55de2aac6"), "node_id" : "11", "root" : "1", "value" : "-79.8868138", "key" : "longitude", "parent" : "1", "children" : nu
ll }
{ "_id" : ObjectId("57317d827db16ab55de2aac7"), "node_id" : "12", "root" : "1", "value" : "3.5", "key" : "stars", "parent" : "1", "children" : null }
{ "_id" : ObjectId("57317d827db16ab55de2aac8"), "node_id" : "13", "root" : "1", "value" : "40.3505527", "key" : "latitude", "parent" : "1", "children" : null
 }
{ "_id" : ObjectId("57317d827db16ab55de2aac9"), "node_id" : "14", "root" : "1", "value" : "map2", "key" : "attributes", "parent" : "1", "children" : [ "15",
"16", "17", "18", "19" ] }
{ "_id" : ObjectId("57317d827db16ab55de2aaca"), "node_id" : "15", "root" : "1", "value" : "True", "key" : "Happy Hour", "parent" : "14", "children" : null }
{ "_id" : ObjectId("57317d827db16ab55de2aacb"), "node_id" : "16", "root" : "1", "value" : "True", "key" : "Accepts Credit Cards", "parent" : "14", "children"
 : null }
{ "_id" : ObjectId("57317d827db16ab55de2aacc"), "node_id" : "17", "root" : "1", "value" : "True", "key" : "Good For Groups", "parent" : "14", "children" : nu
ll }
{ "_id" : ObjectId("57317d827db16ab55de2aacd"), "node_id" : "18", "root" : "1", "value" : "False", "key" : "Outdoor Seating", "parent" : "14", "children" : n
ull }
{ "_id" : ObjectId("57317d827db16ab55de2aace"), "node_id" : "19", "root" : "1", "value" : "1", "key" : "Price Range", "parent" : "14", "children" : null }
```

When the inverted index and forward index are finally generated, they can work together to create the final graph output and allow the user to search for words.

Linking and Word Searches:

After successfully logging in, the user has the option to search the database for a particular key, which can be entered in the textbox provided. Searches can be for a single word or a two-word phrase. For a single word search, all the files containing that word will be displayed. For a two-word search, files containing one or both words will be included in the result.

One word searches:

Upon capturing the user's input, the system queries the 'InvertedIndex' table of MongoDB described in the previous section, and, if the word exists, obtains the list of node ID's containing this word (as a value). For each of the node ID's obtained, the system then queries the 'node' table, i.e. the forward index table, and, based on the information about the parent node, root nodes, etc., computes the path between a node ID and its corresponding root. Finally, upon obtaining the root, the system then constructs the graph of the entire document using the forward index table.

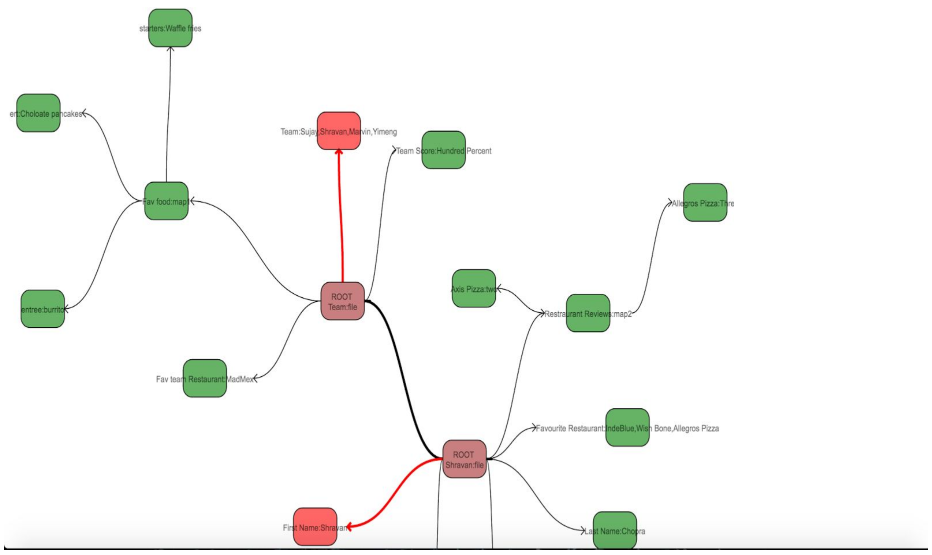
Two word searches:

For each of the two words, the system queries the inverted index table and obtains the set of node ID's for each word. The system then takes the union of the obtained sets, and begins computing the paths from the node ID's to the roots. As with one-word searches, the system constructs the corresponding document trees using the roots. It then dynamically adds links the roots of all the documents obtained in the result.

Rendering the graph:

Upon getting the node ID's from the inverted index, the paths from each node ID to the corresponding roots were obtained, and the system kept track of both the roots as well as all intermediate nodes contained in these paths. For each root, the system would construct the corresponding graph, making sure that the same graph is not constructed twice. While constructing the graph, every intermediate path node (along with the root and the leaf node containing the search term), was colored red, while every other node was colored green. As a design choice, the system queried the entire forward index table just once and stored it locally. This was done in order to prevent the (inevitably) large number of lookups which would otherwise have been made at each step of graph construction and pathfinding, both of which used the forward index table extensively.

Here is a sample graph, representing the search for the word ‘shravan’

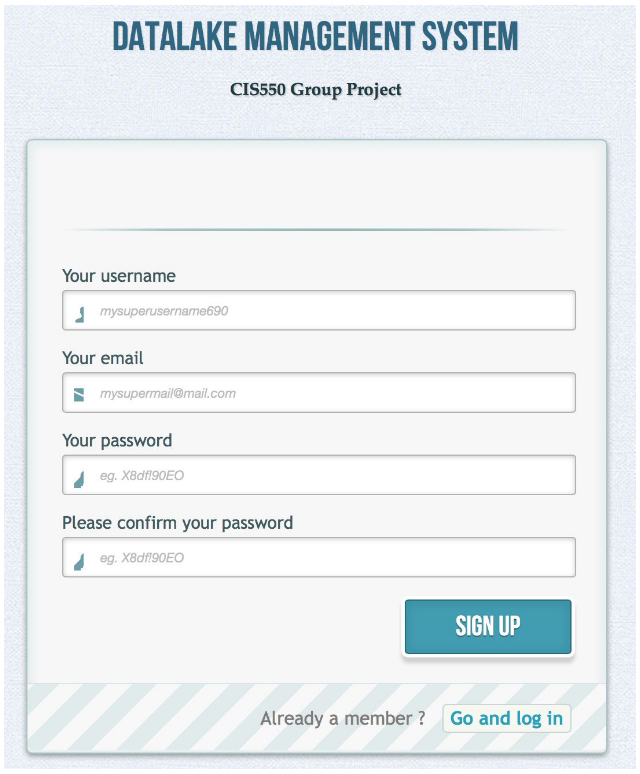


Once the graph is created, we can then output it with the front end side.

Front End:

The basic functions of our front-end include user login page, user sign up page, search page.

Example of the User login and Sign up page:



DATALAKE MANAGEMENT SYSTEM

CIS550 Group Project

Your username

mysuperusername690

Your email

mysupermail@mail.com

Your password

eg. X8df!90EO

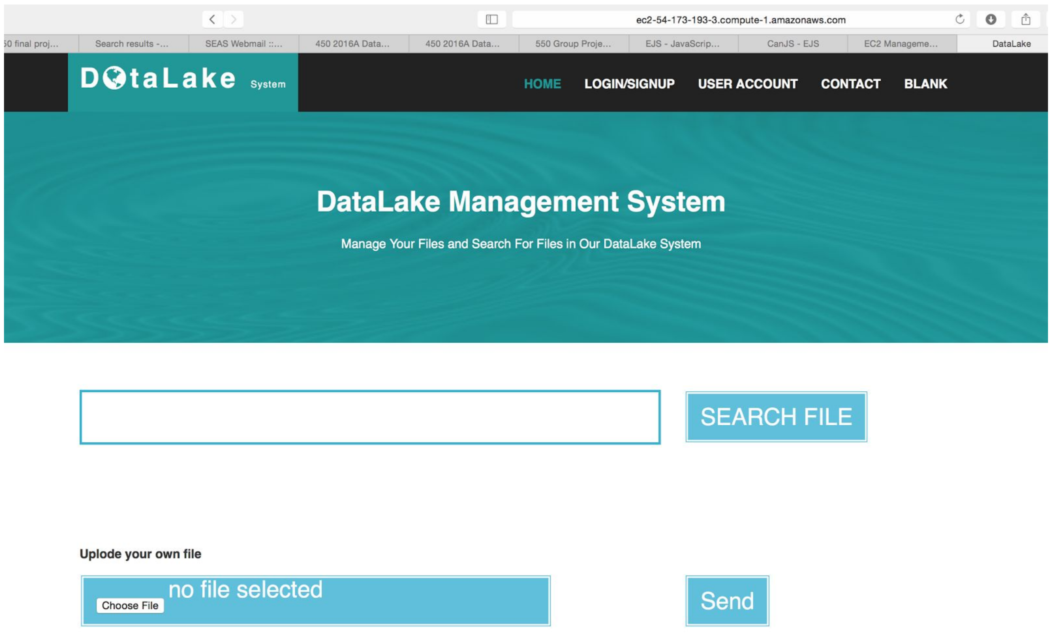
Please confirm your password

eg. X8df!90EO

SIGN UP

Already a member ? Go and log in

Example of the front end search page:


ec2-54-173-193-3.compute-1.amazonaws.com
DataLake System
HOME
LOGIN/SIGNUP
USER ACCOUNT
CONTACT
BLANK
DataLake Management System
Manage Your Files and Search For Files in Our DataLake System
SEARCH FILE
Uplode your own file
Choose File
no file selected
Send

Linking between two pages

The linking between two pages is through the post/get callback. The typical workflow is like this: whenever the user make a post/get from client side, our app.js would get the post/get action, then our app.js call the corresponding functions exports by corresponding separate server side node.js file.

Interaction between client end and server end

Unlike the traditional application, we did not use HTML internal javascript/jquery to realize the user interaction. The reason is that the what we have in our server side, including the MySql, MongoDB, AWS S3 could not be accessed directly by client side for security consideration. A better choice is letting the user post user information from client end to server, and then when server get the information from bodyParser, server could connect with databases and do searching with the information.

To display the feedback information to client side based on server query result, we pass a message to EJS when we render the new page, and client side EJS display the corresponding feedback information out to guide user to do next step. Through that way we could do communication.

Basic backend - MySql user information and permission

In the backend of our system, we use AWS RDS MySql service to hold our user and file information. We have two table UserInfo and FileInfo, UserInfo table stores the user account information, FileInfo table keeps track of the information of the files information when user upload it. Here is the schema of our MySql tables:

UserInfo(<u>username: string</u>, email:string, password:string)

FileInfo(username:string, <u>filename:string</u>, permission:string)

UserInfo table contains all the input informations of the users, which has a primary key username, the FileInfo's entry is inserted whenever the user upload the file, when user upload a file and specify the permission of that file, our system store this information for future search use.

Here is an example workflow of how our server communicate with the server when the user ***sign up***:

- Create the connection pool which allow maximum 100 users to connect at the same time.
- When get the sign up post, get a connection from the pool and check if connected
- If connected, query for the input username, if already exists, re-render the page and output the user has already exist.
- If not exist, insert the post to UserInfo table

Here is an example workflow of how our server communicate with the server when the user ***Log in***:

- Create the connection pool.
- When get the login post, get a connection from the pool and check if connected.
- If connected, query for if the user exist by user name , if not, re-render the page let user sign up first.
- Check if the user password match, if it is, assign the session to username and jump to search page.

Notice:

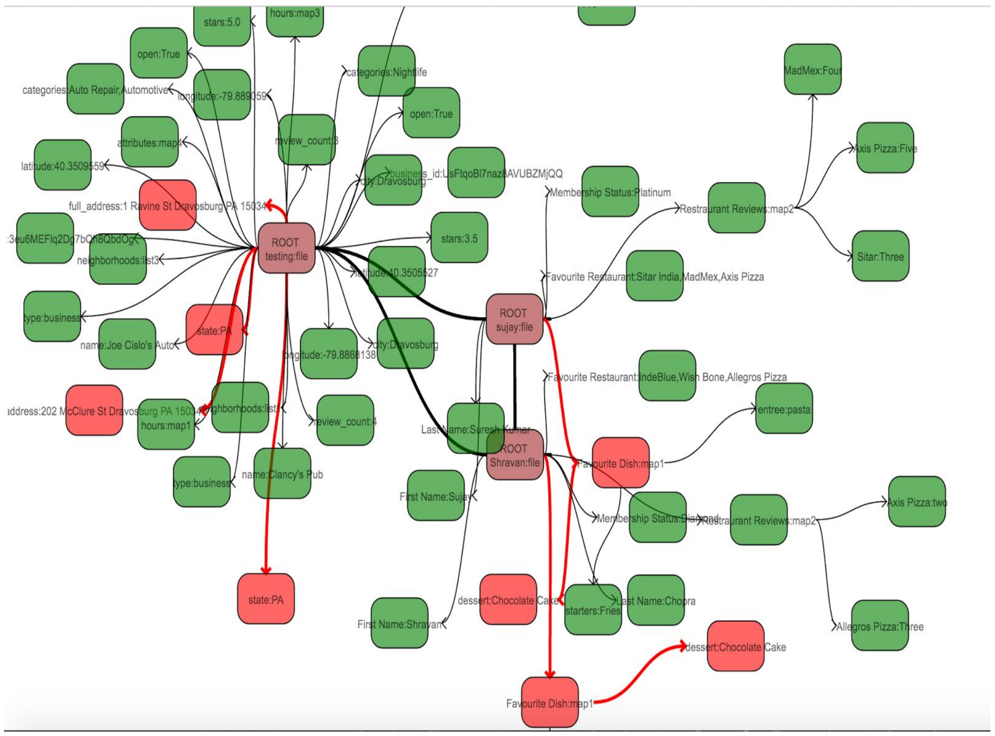
Here we use the connection pool to allow multiple users access the page at the same time. Now we specify maximum 100 connection for the pool, which means we allow at most 100 users to access the same time. To allow the user permission, we require client-sessions and save it once the user login.

Validation of Effectiveness:

Our experiment starts off with simply populating mongodb with many converted json files. This will simulate having many files uploaded into S3 and since the creation of our graphs and search results depend on the mongodb information, we will push in a large number of converted json files. For each of the experiments below, we searched for a single word or a two words. We verified that these are correct based on the requirements described in the “Linking and Web Searches” section of the implementation section. All of these searches returned in a prompt fashion

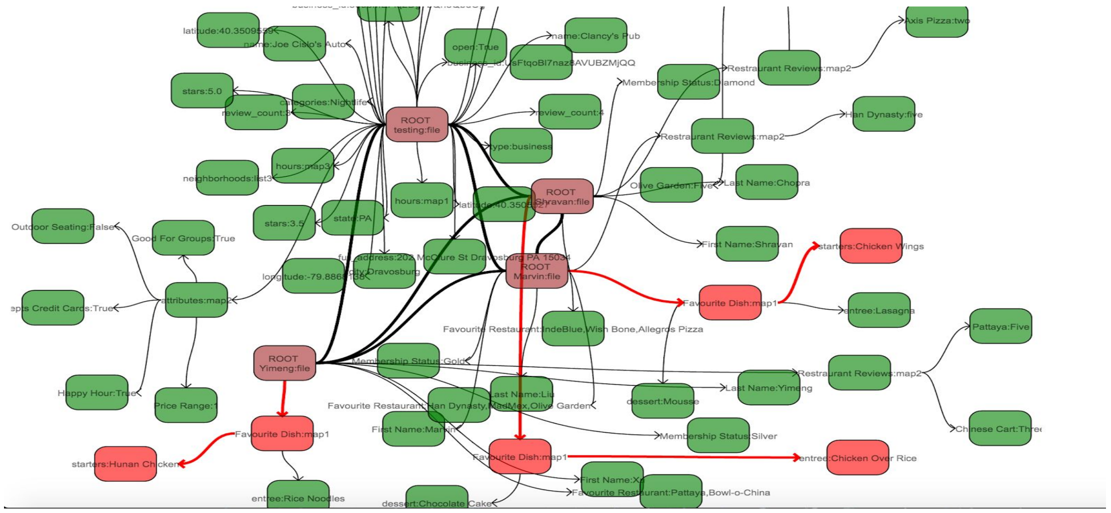
1) Search key: “Chocolate PA”

We are choosing to search for “Chocolate PA” because we wanted to search for related words. For example, in this specific example, we want to search for any places in Pennsylvania that have to do with chocolate. This will create a graph that shows which files give us this information.



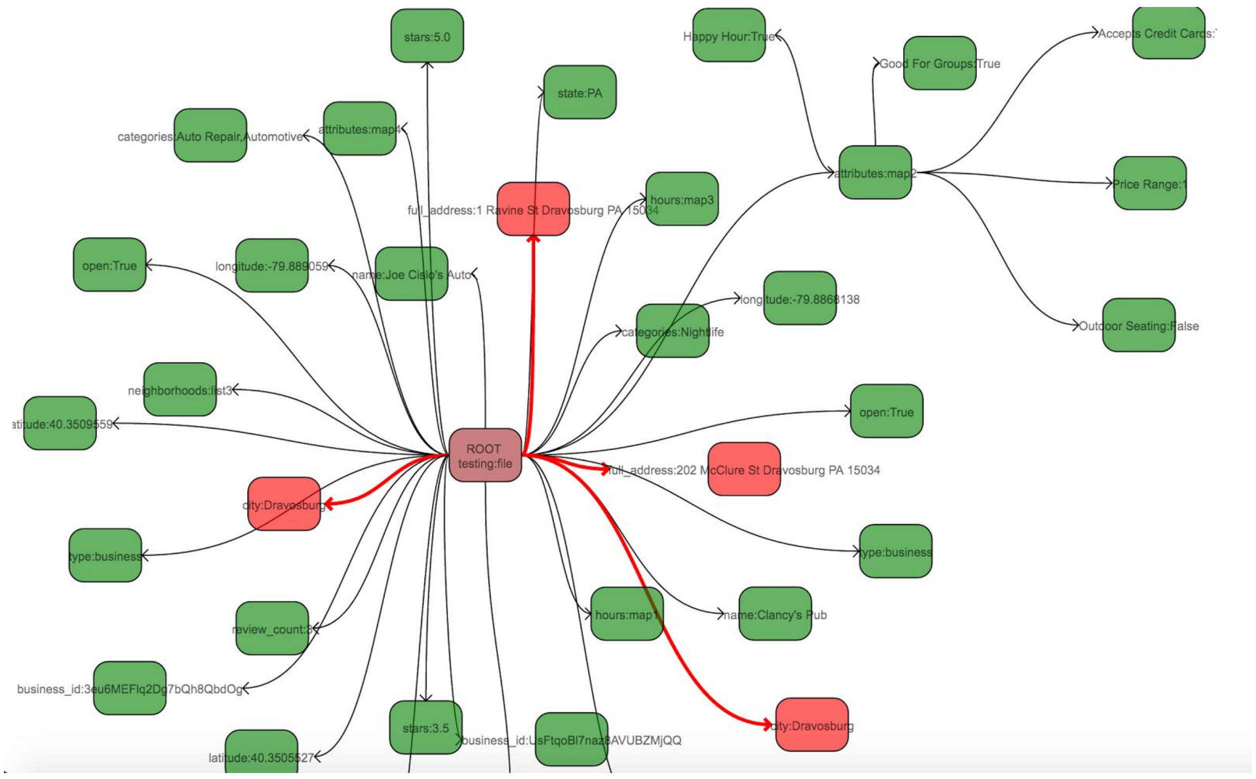
2) Search key: “testing chicken”

We are choosing to search for “testing chicken” because we wanted to search for completely unrelated words. The graph should show the paths that have the testing filename as well as any values that have chicken in them.



3) Search key: “Dravosburg”

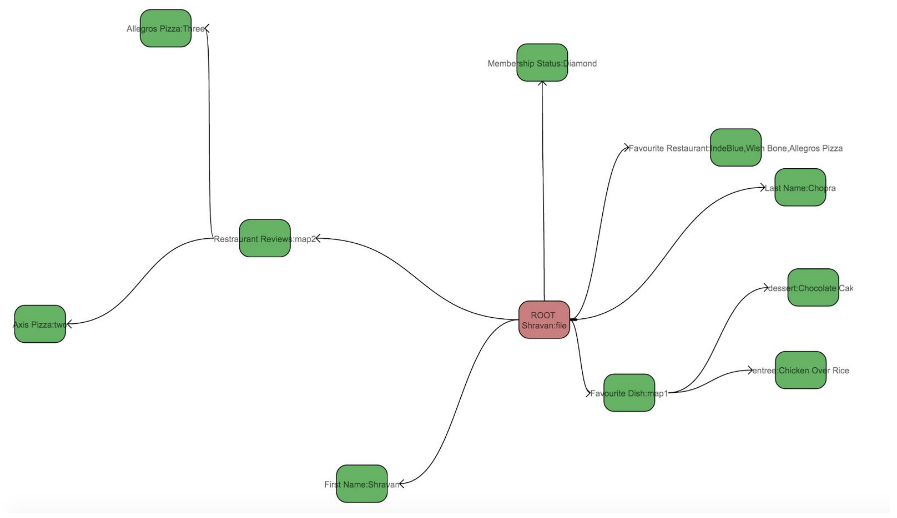
We chose to search for “Dravosburg” because this exists as part of an address value and as its own value. This search should output all nodes that include the word “dravosburg” even if it is a part of an address.



ROOT testing:file
city:Dravosburg
full_address:1 Ravine St Dravosburg PA 15034
full_address:202 McClure St Dravosburg PA 15034
city:Dravosburg

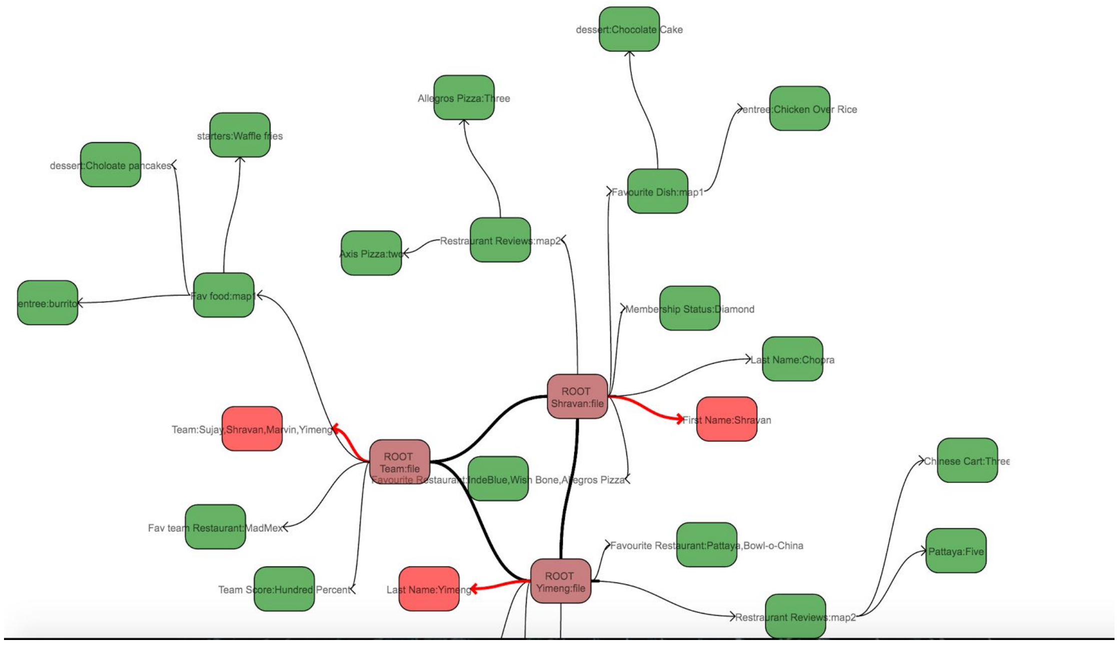
4) Search key: “Shravan”

We chose to search for “Shravan” because we know that this exists as a filename, as well as a node value inside the file “Shravan”. The graph that is output should only provide the node of the “Shravan” file because the value “shravan” exists in the file already.



5) Search key: “yimeng shravan”

We chose to search for “yimeng shravan” because we want to search for specific people who match this search key. The graph that is output should provide us with any nodes that are related to the names of “yimeng” or “shravan”.



Member Contributions:

Marvin: worked on creating the Inverted Index and the Forward Index from the converted JSON files so that the graph can be created and then searched. Also worked on integrating the back end of the project (mongodb, mysql, ec2) with the front end of the project (html/css, javascript, express).

Yimeng: worked on front-end pages with ejs(html)/css/javascript , back-end user information and user permission, MySql connection and query. Also worked on establishing the main framework of the Web application with Node.js Express (both server side and client side). Github repository establishment, AWS RDS database establishment, EC2 instance establishment and connection.

Sujay: worked on the parsers to generate the converted JSON files for every file that is being uploaded into the DLMS. The JSON files have node id's for every value attribute in the document. Also worked on generating the forward index and integrating the backend(multer, file upload-download, mongodb) of the project with the front end.

Shravan: worked on implementing one and two-word searches and dynamic linking. Captured user input from the client side to the server side, queried MongoDB to look for the search term(s) in the inverted index, and used the forward index to construct document trees and illustrate paths to every occurrence of the search term(s). Graphs were rendered using the Dracula Graph Library for JavaScript.